Predicting Boston Housing Prices using Random Forest and Gradient Boosting Models

—

By Gurjot Singh

INTRODUCTION

The Boston Housing Dataset is a popular dataset widely used for training machine learning models in predictive analysis. In this article, I have documented my journey as I navigate through the Boston Housing datasets, applying various techniques to build and evaluate predictive models. I'll focus on mainly two models: Linear Regression and Random Forest and I will also include my references and code snippets.

About Boston Housing Dataset

I came across the dataset while browsing through Kaggle, the dataset is a collection of data related to housing in various neighborhoods in Boston. It has 505 unique values and 13 features and a target value. The target value is "medv", median value of the homes, in thousands of dollars.

Here's a breakdown of the features:

- **CRIM**: Per capita crime rate by town.
- **ZN**: Proportion of residential land zoned for lots over 25,000 sq. ft.INDUS: Proportion of non-retail business acres per town.

-
- **CHAS**: Charles River dummy variable (1 if tract bounds river; 0 otherwise).
- **NOX:** Nitric oxide concentration (parts per 10 million).
- **RM**: Average number of rooms per dwelling.
- **AGE**: Proportion of owner-occupied units built before 1940.
- **DIS**: Weighted distances to five Boston employment centers.
- **RAD**: Index of accessibility to radial highways.
- **TAX**: Full-value property tax rate per $10,000.
- **PTRATIO**: Pupil-teacher ratio by town.
- **B:** 1000(Bk - 0.63)^2 where Bk is the proportion of Black residents by town.
- **LSTAT**: Percentage of lower status of the population.

Data Analysis

I started my analysis to understand the relationship between features and the target values. It included plotting heatmaps and generating a correlation matrix to understand how different features correlate with the housing prices.

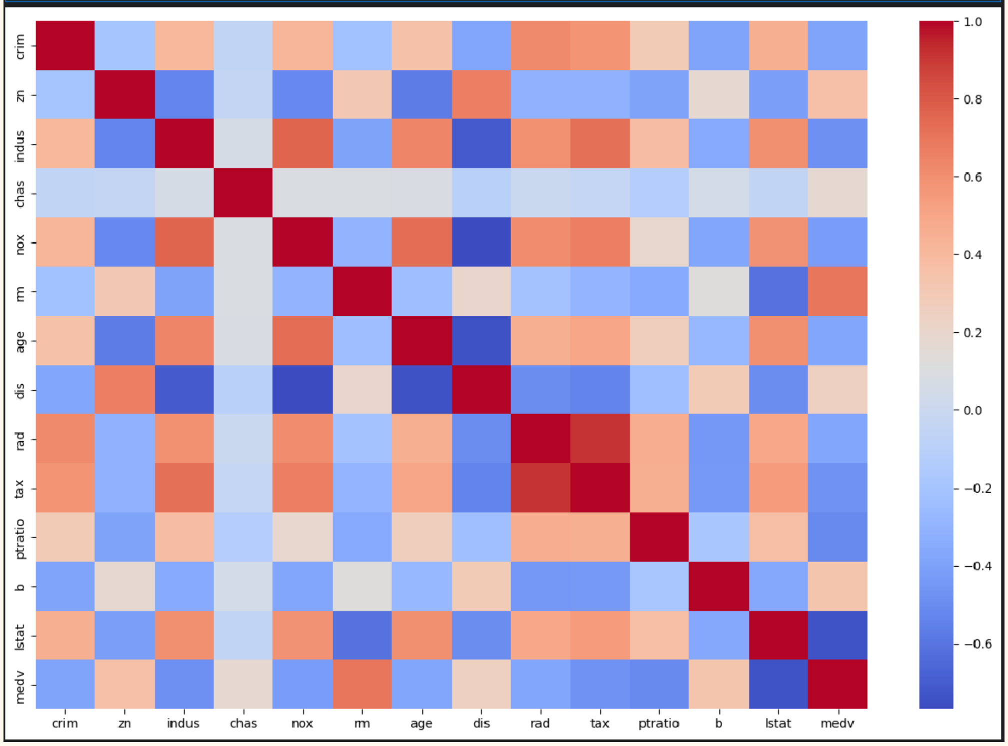
After defining the data frame(df), I created a correlation matrix to assess the relationship between features and the target value.

```
[6]: corr_matrix = df.corr()

[7]: corr_matrix
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	b	lstat	
crim	1.000000	-0.200469	0.406583	-0.055892	0.420972	-0.219433	0.352734	-0.379670	0.625505	0.582764	0.289946	-0.385064	0.455621	-0.
zn	-0.200469	1.000000	-0.533828	-0.042697	-0.516604	0.311173	-0.569537	0.664408	-0.311948	-0.314563	-0.391679	0.175520	-0.412995	0.
indus	0.406583	-0.533828	1.000000	0.062938	0.763651	-0.394193	0.644779	-0.708027	0.595129	0.720760	0.383248	-0.356977	0.603800	-0.
chas	-0.055892	-0.042697	0.062938	1.000000	0.091203	0.091468	0.086518	-0.099176	-0.007368	-0.035587	-0.121515	0.048788	-0.053929	0
nox	0.420972	-0.516604	0.763651	0.091203	1.000000	-0.302751	0.731470	-0.769230	0.611441	0.668023	0.188933	-0.380051	0.590879	-0
rm	-0.219433	0.311173	-0.394193	0.091468	-0.302751	1.000000	-0.240286	0.203507	-0.210718	-0.292794	-0.357612	0.128107	-0.615721	0.
age	0.352734	-0.569537	0.644779	0.086518	0.731470	-0.240286	1.000000	-0.747881	0.456022	0.506456	0.261515	-0.273534	0.602339	-0.
dis	-0.379670	0.664408	-0.708027	-0.099176	-0.769230	0.203507	-0.747881	1.000000	-0.494588	-0.534432	-0.232471	0.291512	-0.496996	0.
rad	0.625505	-0.311948	0.595129	-0.007368	0.611441	-0.210718	0.456022	-0.494588	1.000000	0.910228	0.464741	-0.444413	0.488676	-0.
tax	0.582764	-0.314563	0.720760	-0.035587	0.668023	-0.292794	0.506456	-0.534432	0.910228	1.000000	0.460853	-0.441808	0.543993	-0.
ptratio	0.289946	-0.391679	0.383248	-0.121515	0.188933	-0.357612	0.261515	-0.232471	0.464741	0.460853	1.000000	-0.177383	0.374044	-0.
b	-0.385064	0.175520	-0.356977	0.048788	-0.380051	0.128107	-0.273534	0.291512	-0.444413	-0.441808	-0.177383	1.000000	-0.366087	0.
lstat	0.455621	-0.412995	0.603800	-0.053929	0.590879	-0.615721	0.602339	-0.496996	0.488676	0.543993	0.374044	-0.366087	1.000000	-0.
medv	-0.388305	0.360445	-0.483725	0.175260	-0.427321	0.696169	-0.376955	0.249929	-0.381626	-0.468536	-0.507787	0.333461	-0.737663	1.0

Next, for better understanding, I decided to plot a heatmap of the corr_matrix using the seaborn and matplotlib library.

```
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize = (15,10))
sns.heatmap(corr_matrix, cmap = "coolwarm")
plt.show()
```



Here:

-Red represents positive correlations which means that when one feature increases the other also tends to increase, whereas blue represents negative correlation.

- I saw that rm, avg room per dwelling and medv have the strongest correlation, which suggest that houses with more rooms have higher values.

- There is also a strong negative correlation between lstat, percent of lower-status population and medv.

In addition to this, I calculated a correlation matrix and got the correlations with the target, medv, which also gave similar results.

```
#correlation to the median house value
medv_corr = corr_matrix["medv"].sort_values(ascending = False)
print(medv_corr)
```

```
medv       1.000000
rm         0.696169
zn         0.360445
b          0.333461
dis        0.249929
chas       0.175260
age       -0.376955
rad       -0.381626
crim      -0.388305
nox       -0.427321
tax       -0.468536
indus     -0.483725
ptratio   -0.507787
lstat     -0.737663
Name: medv, dtype: float64
```

Creating test and train sets

The next step was to create a test set for which I used the StratifiedShuffleSplit from the sklearn.model_selection library. I did this to get rid of any sampling bias in my testing and training sets, this technique helps in maintaining a balanced representation of each category.

The first step was to create a new categorical attribute "medv_cat" by dividing "medv" values into 4 bins, using the pd.cut.

```
[31]: df["medv_cat"] = pd.cut(df['medv'],
                        bins= [0,15,25,35,50],
                        labels= [1,2,3,4])
```

Now it was time to apply the StratifiedShuffleSplit to create the sets.

```
[34]: from sklearn.model_selection import StratifiedShuffleSplit
      import numpy as np

      split=  StratifiedShuffleSplit(n_splits = 1 , test_size = 0.2 , random_state = 42)
      for train_index, test_index in split.split(df, df["medv_cat"]):
          strat_train_set = df.loc[train_index]
          strat_test_set = df.loc[test_index]
```

I created the "medv_cat" attribute for the purpose of ensuring a balanced split, now it is no longer needed. Therefore, I removed it to ensure that the training and test set only have the original features and the target value.

```
[37]: for set_ in (strat_train_set , strat_test_set):
          set_.drop('medv_cat' , axis = 1, inplace = True)
```

Now the next step is to separate features (input data) from the target variable (the value we are trying to predict), which is essential for machine learning.

Here is the snippet I used:

```
[49]: # Separate features and target for training set
      X_train = strat_train_set.drop("medv", axis=1)  # Features
      y_train = strat_train_set["medv"]  # Target

      # Separate features and target for test set
      X_test = strat_test_set.drop("medv", axis=1)  # Features
      y_test = strat_test_set["medv"]  # Target
```

-X_train and X_test contain all the features, excluding the target value (medv).

-y_train and y_test contain the target value.

This ensures that the model uses the features during training and evaluation, while the target variable is kept aside for prediction and accuracy assessment.

Model and Evaluation

I applied various machine learning models to the dataset, namely Linear regression, Random Forest and Gradient Boosting. For each of these models RMSE (Root Mean Squared Error) was calculated, which measures the average difference between the actual values and values that my model predicts. A lower RMSE value indicates better model performance.

Linear Regression:

This was used as a baseline to compare the performance of other models. This snippet was used to train the model on the training set and then the RMSE was calculated (sklearn was used here).

```
[54]: from sklearn.linear_model import LinearRegression
      lr = LinearRegression()

[57]: lr.fit(X_train, y_train)

[57]: ▾ LinearRegression
      LinearRegression()

[58]: pred1 = lr.predict(X_test)

[60]: from sklearn.metrics import mean_squared_error

      mse = mean_squared_error(y_test , pred1)
      rmse = mse**0.5
      print(f"RMSE: ", rmse)

      RMSE:  4.789080398339908
```

RMSE = 4.789080

Random Forest:

Next, I implemented a Random Forest Regressor which combines the predictions of many decision trees to reduce overfitting and improve accuracy.

```
[64]: from sklearn.ensemble import RandomForestRegressor
      forest = RandomForestRegressor()

[66]: forest.fit(X_train , y_train)

[66]: ▾ RandomForestRegressor
      RandomForestRegressor()

[67]: pred2 = forest.predict(X_test)

[69]: mse_rf = mean_squared_error(y_test, pred2)
      rmse_rf = mse_rf**0.5

      print("Random Forest RMSE : ", rmse_rf )

      Random Forest RMSE :  3.244146698890416
```

RMSE = 3.2441466

Hyperparameter Tuning for Random Forest :

While Random Forest initially performed well, I wanted to further improve its accuracy by tuning key hyperparameters. Two important hyperparameters in Random Forest are:

- n_estimators: The number of trees in the forest. More trees usually result in better performance but increase computation time.
- max_depth: The maximum depth of the trees. Deeper trees capture more complex patterns but may overfit the data.

To optimize these parameters, I used GridSearchCV, which tries different combinations of hyperparameters and selects the one with the best performance.

```
from sklearn.model_selection import GridSearchCV
```

```
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth':[10,20,30]
    }

grid_search = GridSearchCV(forest, param_grid, cv = 5, scoring= 'neg_mean_squared_error' )
grid_search.fit(X_train, y_train)
print(grid_search.best_params_)
```

```
{'max_depth': 30, 'n_estimators': 100}
```

```
rf_model = RandomForestRegressor(n_estimators = 100, max_depth = 30)
rf_model.fit(X_train, y_train)
```

```
RandomForestRegressor
RandomForestRegressor(max_depth=30)
```

```
new_pred = rf_model.predict(X_test)
mse_rf_ = mean_squared_error(y_test, new_pred)
rmse_rf_ = mse_rf_**0.5

print(f"Random Forest RMSE: {rmse_rf_}")
```

```
Random Forest RMSE: 3.183419709843428
```

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error

gbr = GradientBoostingRegressor(n_estimators=100, max_depth=3, random_state=42)
gbr.fit(X_train, y_train)

y_pred_gbr = gbr.predict(X_test)

mse_gbr = mean_squared_error(y_test, y_pred_gbr)
rmse_gbr = mse_gbr ** 0.5

print(f"Gradient Boosting RMSE: {rmse_gbr}")
```

```
Gradient Boosting RMSE: 3.173850838695141
```

This process helped me find the best combination: n_estimators = 100 and max_depth = 30. I retrained the Random Forest model using these values, which resulted in an RMSE of 3.18. After tuning Random Forest, I also tested a Gradient Boosting Regressor, which yielded an RMSE of 3.17, performing slightly better.

Saving the models : To save these models, I used joblib.

```
[76]: import joblib

      joblib.dump(lr ,"linear_regression.pkl")
      joblib.dump(forest, "forest_regression.pkl")
```

Conclusion

In this project, I used the Boston Housing Dataset to predict housing prices using various machine learning models

- **Linear Regression**, while a simple baseline model, had the highest RMSE of **4.78**.
- **Random Forest**, with its ability to handle nonlinear relationships and interactions between features, performed significantly better with an RMSE of **3.24**.
- After **hyperparameter tuning** using **GridSearchCV**, the performance of Random Forest improved slightly, achieving an RMSE of **3.18**.
- The **Gradient Boosting Regressor**, another ensemble method, performed the best, with an RMSE of **3.17**, though the improvement over Random Forest was marginal.

The key takeaways are models like Random Forest and Gradient Boosting are highly effective for these regression tasks. Hyperparameter tuning is also crucial in optimizing the performance model.

References

1. **Boston Housing Dataset**
   The dataset used in this project was obtained from Kaggle.
   Link: https://www.kaggle.com/datasets/arunjangir245/boston-housing-dataset

2. **Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow**
   Géron, Aurélien. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 2nd edition, O'Reilly Media, 2019.